2024

Solution Architect BP – Ejercicio Práctico

CEDRIC BARDALEZ HALL

índice

Desarrollo de la Solución

Detalles solicitados

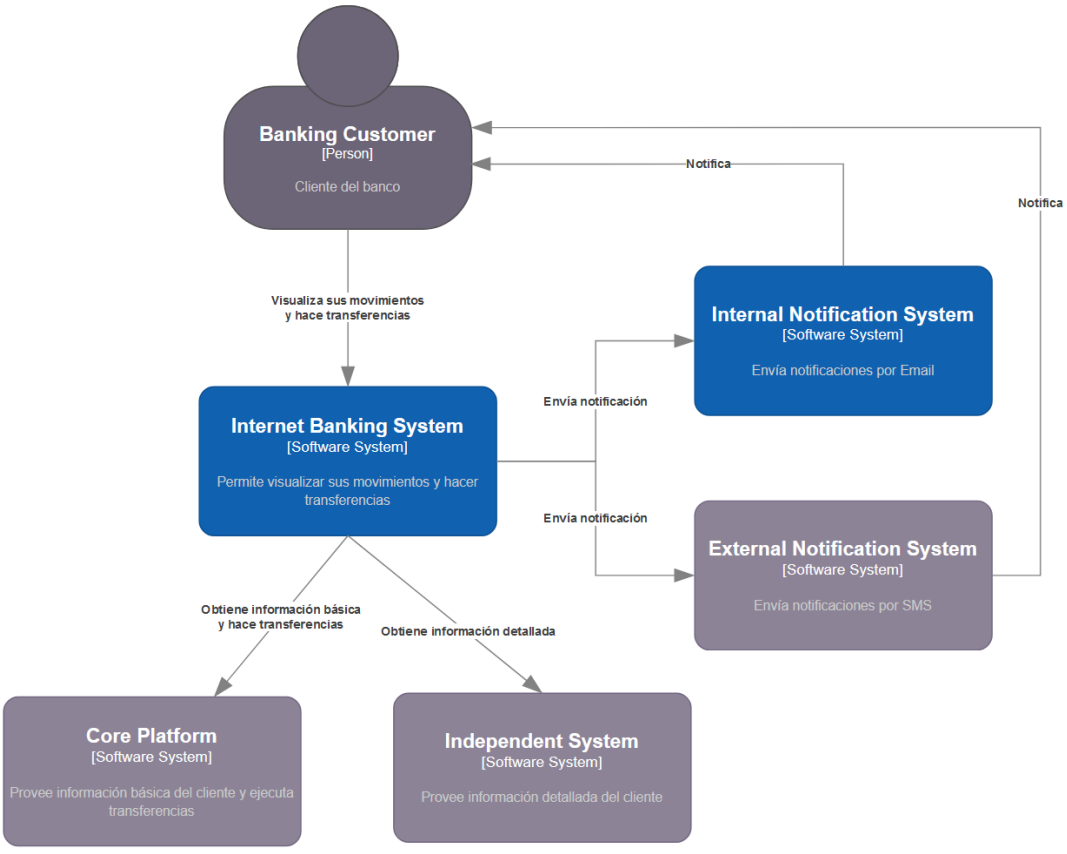
A lo largo de este documento iremos desarrollando la arquitectura correspondiente para un sistema de banca por internet, conforme a los requerimiento que se detallan en las instrucciones del ejercicio.

Se nos informa lo siguiente:

1. Existen dos sistemas previos
    a. **Una plataforma core:** Brinda información básica del cliente, movimientos, productos
    b. **Un sistema independiente:** Brinda información completa del cliente
2. Y se solicita:
    a. El sistema debe contar con 2 aplicaciones de acceso al usuario (una web SPA y una aplicación móvil)
    b. El sistema debe notificar a los usuarios sobre los movimientos realizados
    c. El sistema debe permitir autenticarse siguiendo el protocolo OAuth2
    d. El sistema deberá tener un proceso de onboarding con biometría facial

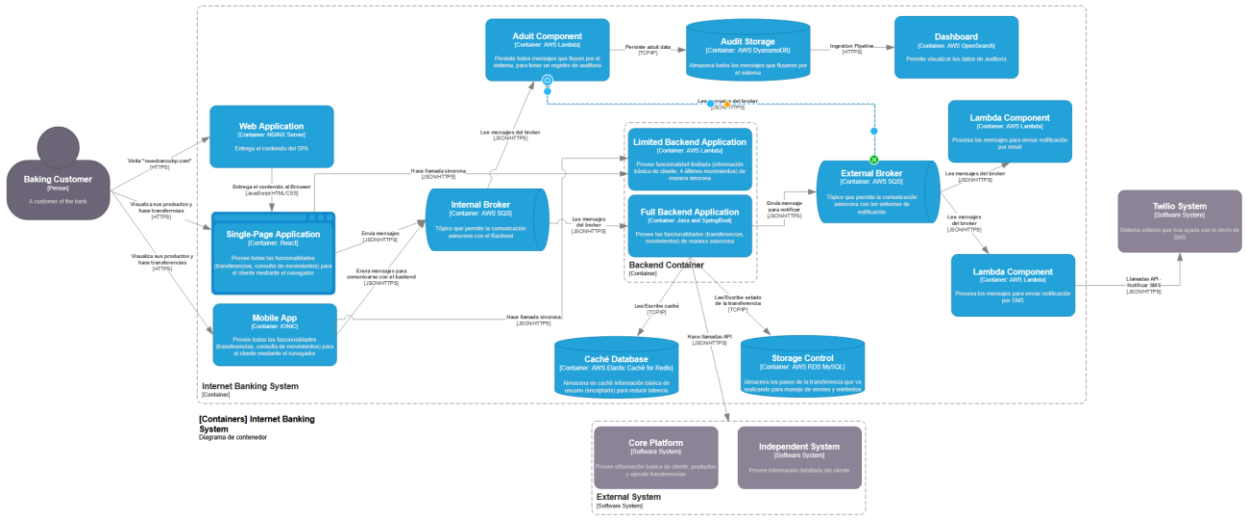
Empecemos diagramando el **diagrama de contexto** de C4

Diagrama de contexto – C1



[System Context] Internet Banking System
Diagrama de contexto

Diagrama de contenedores – C2



Consideraciones sobre el sistema

Observación: Diseñaremos una aplicación muy ligera, que escale a millones de solicitudes por minuto, donde el usuario puede realizar una transferencia y posteriormente se le notificara si la transferencia fue correcta o no, de igual forma al solicitar su reporte se le enviará al correo los movimientos dentro de las fechas solicitadas.

Esta aplicación usará mensajería asincrona y componentes desacoplados, permitiendo una mayor escalabilida; sin embargo, nuestra aplicación tendrá una arquitectura híbrida (sincrona y asincrona), debido a que como todo nuestro procesamiento y respuesta son asíncronas, debemos implementar una servicio síncrono que nos retorne respuestas rápidas y ligeras para que al abrir el aplicativo se pueda mostrar información básica del usuario y también sus últimos movimientos, por tanto, implementaremos una función lambda y un nuevo endpoint en el core platform.

- El nuevo endpoint nos permitirá obtener las ultimas 4 transferencias, a diferencia del endpoint ***obtener movimientos***, éste es mucho más ligero y solo obtiene unos pocos datos (la fecha de transferencia, el monto y el nombre de destino).
- Y una función lambda que retornará la información del usuario y/o sus 4 últimas transferencias según lo solicitemos

Diagrama de despliegue (Implementación en AWS)

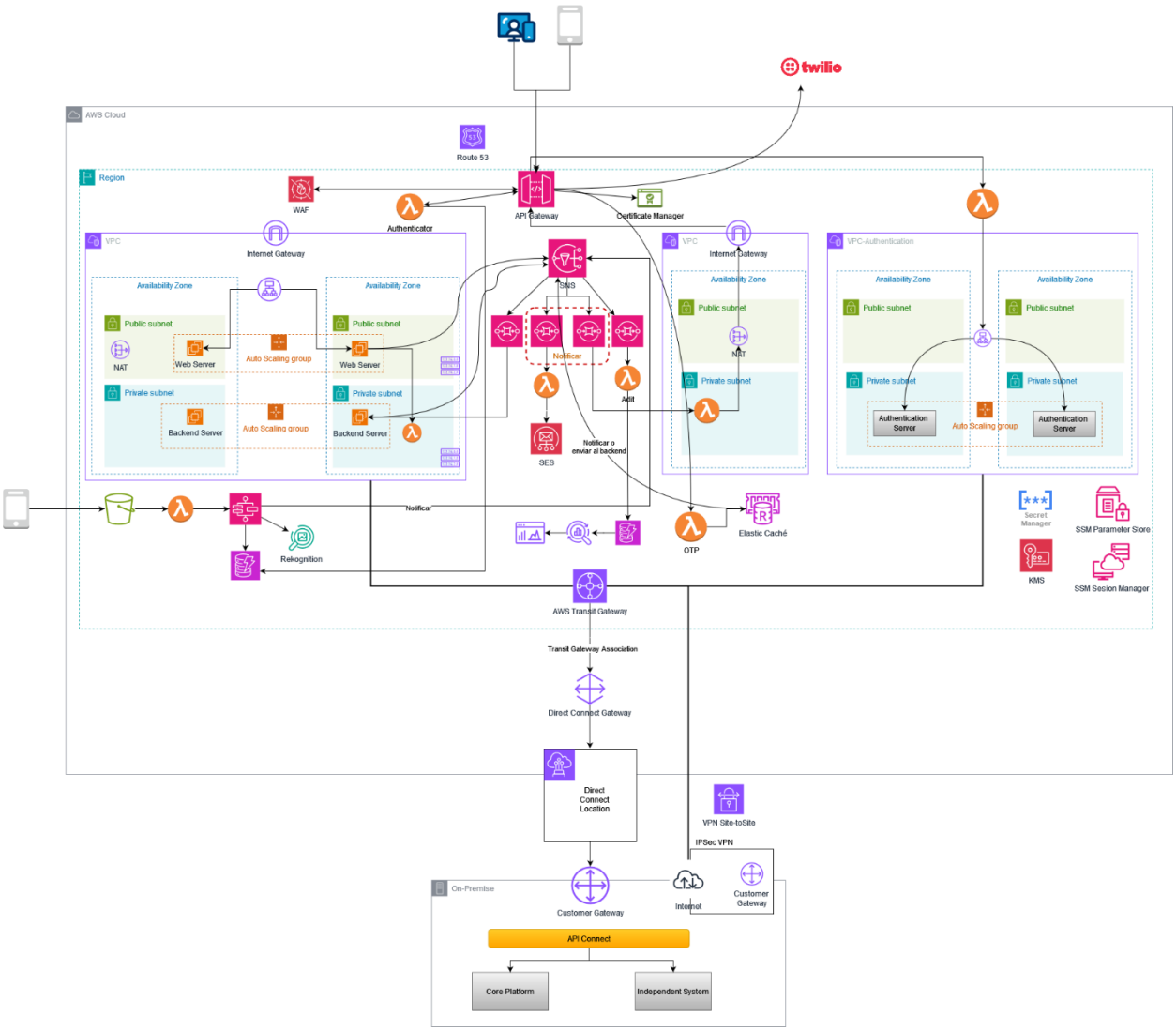
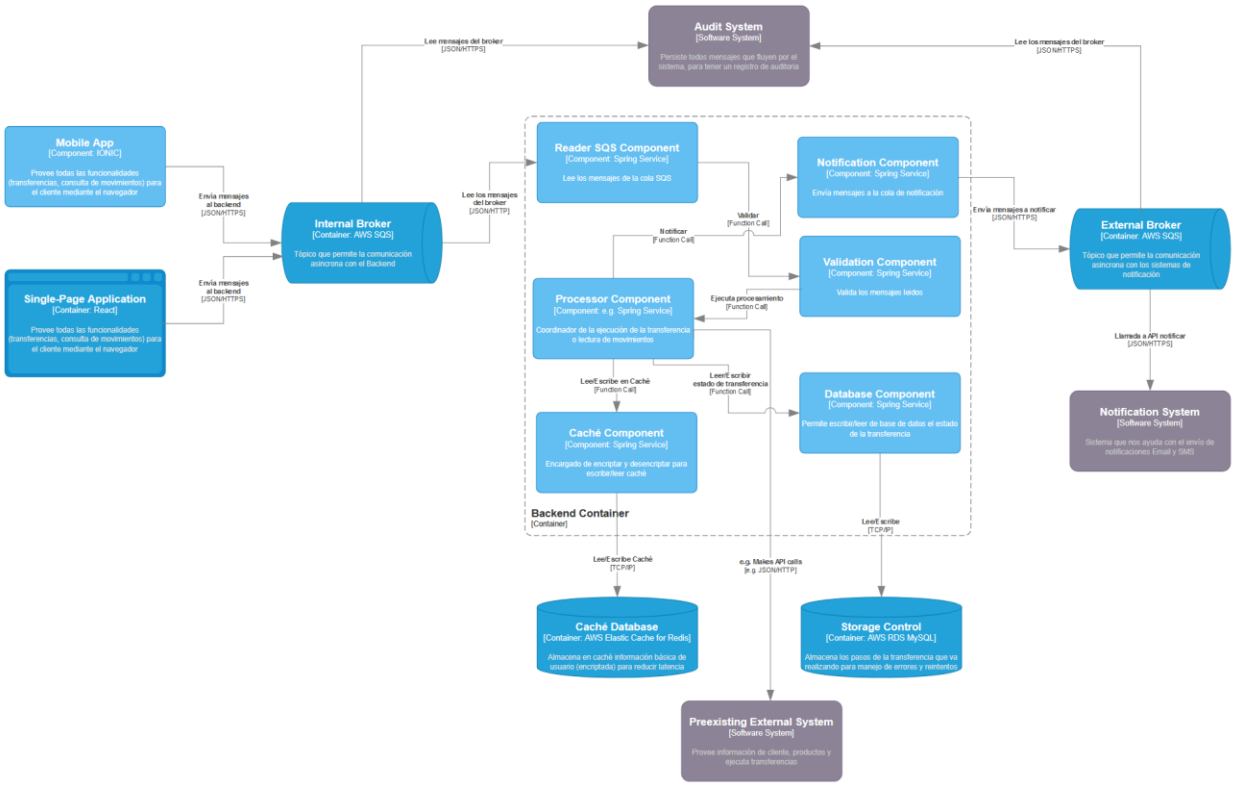


Diagrama de componentes – C3

No queremos tener un diagrama de componentes C3 muy sobrecargado y prácticamente ininteligible. Notemos como nos quedó el diagrama anterior (despliegue en AWS) y rápidamente vemos que es complejo, por tanto, hemos optado por segregar el diagrama en varias partes, las cuales se irán desarrollando a lo largo del documento, tales como “Sistema de notificación”, “Auditoria”, “Autenticación”, “Onboarding”, etc.

En este diagrama únicamente nos centraremos en los componentes del container backend (haciendo “zoom” sobre ellos)



Herramientas para la aplicación (Web y Móvil)

Comparativa de herramientas para el cliente Web

Para el desarrollo del frontend tenemos como opción algunas librerías o frameworks, dentro de las más conocidas, por su utilización en grandes proyectos, tenemos al framework Angular y la librería React

Cuadro comparativo de trade-off

Tecnología vs Característica	Angular	React
Arquitectura	Basado en MVC, disponemos de componentes, enrutadores y servicios	Basado en una arquitectura Flux (flujo de datos unidireccional)
Rendimiento	Bueno, permite carga Lazy Loading para optimizar el número de archivos enviados	Excelente, implementa Virtual DOM lo cual reduce el número de instrucciones para transformar un DOM en otro
Lenguaje	TypeScript, el cual debe transpilarse	JSX (extensión de JavaScript) que permite incrustar etiquetas XML/HTML en el archivo JavaScript
Curva de aprendizaje	Posee buena documentación, con ejemplos prácticos y una comunidad amplia y fuerte. Sin embargo, la curva se aprendizaje es superior a React	Documentación robusta y buena comunidad. La curva de aprendizaje para React suele ser menos pronunciada
Velocidad de desarrollo	Tiene una CLI bien desarrollada, que ofrece una gran experiencia para crear proyectos y componentes	El uso de bibliotecas de terceros puede afectar la velocidad y productividad
Data Binding	Angular utiliza el enfoque **Two-way-binding** que cambia el estado del modelo automáticamente cuando	React utiliza un enfoque unidireccional. Permite el cambio en los elementos de la interfaz solo

	realiza cambios en los elementos de la interfaz y viceversa	después de que se realiza un cambio en el estado del modelo
Detección de errores	La detección de errores en una plantilla se produce en tiempo de ejecución	La detección de errores se genera en la compilación de la plantilla

Observación: Usar TypeScript puede "forzar" a que el código sea más limpio, ordenado, siguiendo patrones de programación funciona y orienta a objetos, y esto podría a posteriori impactar en la mantenibilidad

Conclusión

En la aplicación solicitada no tenemos un gran número de funcionalidades, al menos en lo requerido inicialmente solo tenemos 3 operaciones básicas, de igual forma no necesitamos hacer uso eficiente de "two-way-binding", ni la modularidad que ofrece angular, por otra parte, tampoco necesitamos los beneficiarnos de un "fast-rendering" de React (ligeramente mejor que Angular), ni tampoco aplicaremos el patrón Redux (en el cual React brilla sobre angular). Es decir, debido a la simpleza de la aplicación es igual usar uno u otro, pero nos inclinaremos por React ya que la curva de aprendizaje es menor que la de Angular

Comparativa de herramientas para el cliente Móvil

Existen múltiples opciones en el mercado tales como Flutter, IONIC, React Native, etc. pero ya que se está trabajando con lenguaje JavaScript para el desarrollo de la aplicación web, sería ideal seguir la misma línea, aprovechando las capacidades ya comprobadas del equipo, no cambiar a otra tecnología que implique capacitaciones, nuevas contrataciones, etc., por tanto, Flutter queda descartado (además, Flutter no suele tener muchos profesionales disponibles en el mercado). Nos queda la difícil decisión entre React Native y IONIC

Cuadro comparativo - IONIC

IONIC	
Ventajas	**Desventajas**
Excelente documentación	Algunas características premium requieren un pago, pero es de código abierto
Curva de aprendizaje baja (ya que está basado en HTML, CSS y JavaScript)	El rendimiento es significativamente menor que una aplicación nativa
Puedes convertir tu aplicación en una PWA con pocos pasos	Las aplicaciones ocupan mayor espacio que las desarrolladas nativamente
Tiene el enfoque de "escribir una vez, ejecutar en cualquier lugar"	Ejecuta la misma interfaz de usuario en todas las plataformas, utilizando diseño web responsivo
Compatible con IOS, Android, Electron y Web	

Cuadro comparativo– React Native

React Native	
Ventajas	**Desventajas**
Posee una gran comunidad y es ampliamente usado en múltiples proyectos	Aunque existe una gran comunidad, muchas bibliotecas no tienen una evolución activa

Genera componentes nativos a partir del JSX,	Soporta oficialmente solo IOS y Android (existen algunos proyectos no oficiales que agregan soporte para sitios web)
Tenemos un resultado tanto de **apariencia como de rendimiento muy similar,** ligeramente inferior a un desarrollo nativo,	Tiene un enfoque de "aprender una vez, escribir en cualquier lugar"
	Los desarrolladores deben crear pantallas de interfaz de usuario específicamente para IOS y para Android.

Finalmente, los puntos más importantes a tomar en cuenta son; el rendimiento y el tiempo de desarrollo (un solo desarrollo para ambos canales – móvil y web), el primero nos brinda una mejor experiencia de usuario y el segundo nos brinda un mejor “time-to-market” y homogeneidad entre canales.

Debido a que la diferencia de rendimientos para una aplicación tan pequeña no es significativa, y como banco deseamos mantener una experiencia de usuario homogénea en los diferentes canales con el menor esfuerzo de desarrollo. Además, hay que tener en cuenta que necesitamos acceder a la cámara para tomar fotos en el proceso de onboarding, y también necesitamos huella dactilar para el proceso de login (son funciones nativas que IONIC nos muestra compatibilidad en su documentación con ejemplos y buen funcionamiento)

Por lo tanto, nos decantamos por IONIC, es decir nuestra elección final sería IONIC y React

Sistema de Notificación

Para notificar podemos utilizar notificaciones simples SMS, principalmente por su rapidez, ya que ellas pueden llegar incluso en momentos de congestión de la red celular, y no necesitan acceso a internet, pero como se requiere doble notificación, también lo haremos por email.

Notificaciones Email:

- Puede formatearse en HTML (hasta 10mb si usamos el servicio AWS SES)
- Son más coloridas y bonitas
- Pueden demorar más en llegar
- El usuario necesita tener acceso a internet para recibirla

Notificaciones SMS

- Texto plano (hasta 256kb si usamos el servicio AWS SNS)
- Son más rápidas
- No necesita tener acceso a internet

Para nuestra solución un sistema externo notificara por SMS (con ayuda de Twilio), mientras el interno lo usaremos para notificar por correo (por medio de AWS SES), es decir el usuario recibirá notificación a su correo y SMS

Diagrama de despliegue (implementación en AWS)

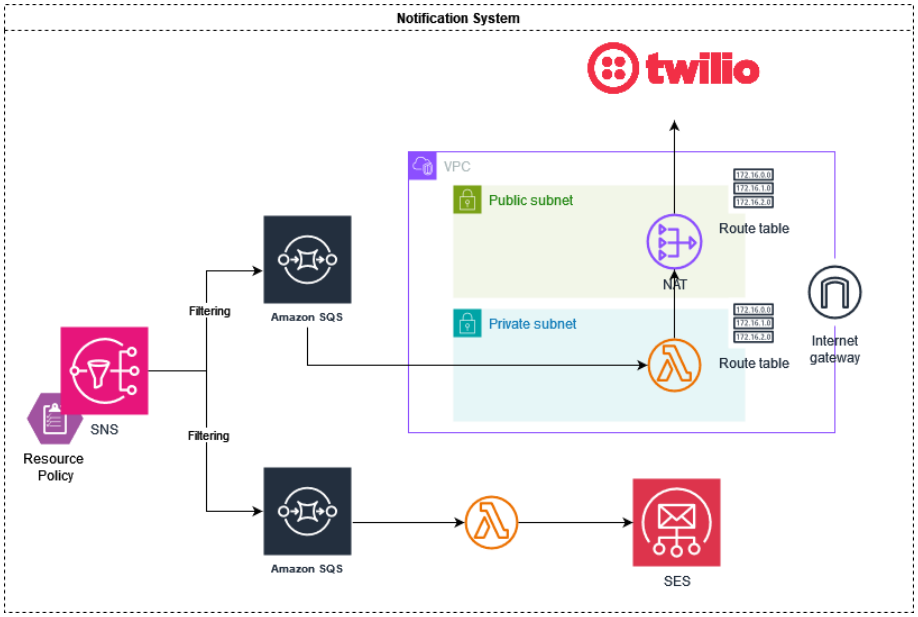
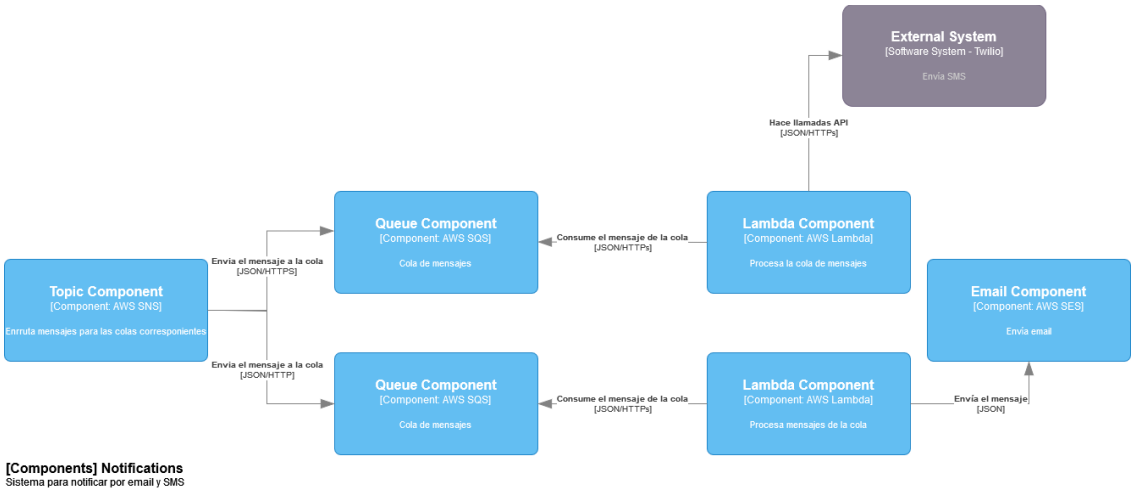


Diagrama de componentes – C3



Descripción:

Nuestro servicio backend enviará un mensaje a un tópico SNS (haciendo uso del patrón Publisher/Subscriber). AWS SNS básicamente lo podemos configurar de 2 formas en modo FIFO o modo estándar. En ninguna parte del ejercicio solicitado nos hablan del volumen de request concurrentes que deberá soportar el sistema, pero supondremos que se trata de un número elevado y como AWS SNS FIFO solo puede soportar 3,000 mensajes por segundo, optaremos por el modo “STANDARD”, lo cual nos lleva a la dificultad que los mensajes podrían no estar en orden y podrían no ser únicos (repetirse).

Cedric Bardalez Hall

Sin embargo, no existe ningún problema en que las notificaciones no sigan un orden exacto ni tampoco sería grave para el negocio que se notifique dos veces la misma operación a un usuario.

Usaremos AWS SNS filtering Body para filtrar si el mensaje debe enviarse por nuestro sistema (cola número 1 - Email), o por un sistema externo (cola número 2 - SMS) o por ambos, por lo tanto, AWS SNS "empujará" el mensaje a las colas AWS SQS y funciones lambda lo consumiran, una de las cuales envía el SMS mediante Twilio y la otra función lambda enviará un correo electrónico apoyándose del servicio AWS SES.

Observación 1: Una de las funciones lambda está adjunta a una subnet privada de otra VPC y requiere salida a internet, para poder hacer un request a Twilio, lo cual logra por medio de un NAT Gateway desplegado en la subnet pública.

Observación 2: Si la función lambda no pudiese entregar el mensaje deberá enviarlo a una cola AWS SQS de análisis (recordemos que el periodo de retención de una cola SQS va desde 1 minuto hasta los 14 días, donde 4 días es la configuración por defecto).

Observación 3: Si el mensaje no puede ser entregado a la función lambda, entonces SQS deberá descartar el mensaje y enviarlo a una cola DLQ.

Observación 4: No podemos permitirnos problemas de **cold star**, sobre todo a altas horas de la madrugada donde no se realizan muchas transacciones y podría estar "apagada" nuestra función lambda, por tanto, podríamos usar ***Provisioned Concurrency*** pero el costo se elevaría notablemente.

Una de las razones principales de usar lambda para notificar y no un servicio que se ejecute 24x7 es debido a que las notificaciones suelen ser de procesamiento corto (menor a 15min) y esporádico, y no tendrían sentido mantener un proceso activo siempre.

Para mantener la economía desarrollaremos las funciones lambda usando el lenguaje Java y activaremos la característica de ***SnapStart*** (disponible solo para runtime Java 11 o superior), lo cual nos permitirá reducir el tiempo de arranque hasta en 10x

Observación 5: Realmente cuando se tiene posible duplicación de mensajes se debería de implementarse la idempotencia, es decir de alguna manera deberíamos logar que las funciones lambda se puedan ejecutar varias veces sin causar duplicados, pero en este caso particular no es necesario realizar tanto trabajo ya que realmente no importa la duplicación

Consideraciones sobre las colas de mensajes

Para desacoplar los componentes utilizaremos mensajería asíncrona y la aplicación Frontend enviará un mensaje a SQS y habrá un grupo de instancias Backend que consumirán ese mensaje.

Consideraciones sobre SNS y SQS

Tendremos que hacer algunas configuraciones en las colas SQS, por ejemplo, debemos activar ***sqsManagedSseEnabled*** para habilitar la encriptación de mensajes del lado del servidor y forzar que las conexiones de los consumidores y productores sean mediante HTTPS.

También debemos configurar un pool de mensajes adecuado, y el tiempo de espera antes de recibir una respuesta de que no hay mensaje disponible mediante el parámetro **reciveMessage**

También debemos configurar el **visibilityTimeOut** para establecer el tiempo en el cual el mensaje vuelva a ser visible, para otros consumidores si no fue procesado exitosamente

Proceso de autenticación

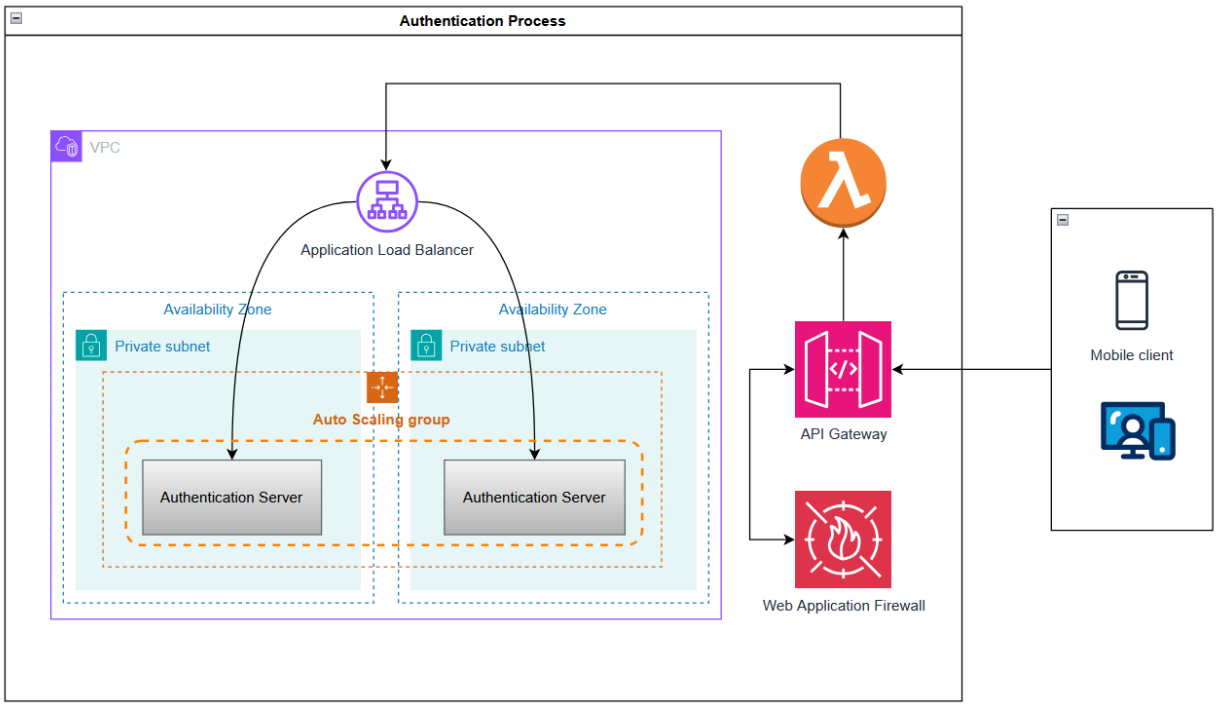
Sabemos que OAuth2 es un protocolo de autenticación y no de autorización, este tiene 5 principales "flujos" (grant_type)

- Authorization Code
- Implicit
- Client Credentials
- Password
- Refesh Token

Se nos dice que ya existe un componente que hace dicho proceso de autenticación e implementa el protocolo OAuth2, por lo tanto, escogeremos el flujo de Password. Ya que tanto nuestra aplicación web como móvil son terceros de confianza (se trata del mismo aplicativo bancario), autenticaremos nosotros, no necesitamos un acceso federado, ni tampoco necesitamos manejar un código de autorización para cambiarlo por un token y hacer request sobre un servidor de recursos externo cuyos recursos que son propiedad del usuario final.

Simplemente necesitamos que la aplicación web o móvil actúe directamente en nombre del usuario quien se autentica con MFA para verificar que sea quien dice ser. Manejaremos un token JWT

Diagrama de despliegue (implementación en AWS)



Cedric Bardalez Hall

Para el ingreso con Usuario y Password se deriva la solicitud directamente al componente que ya existe (server UAA), si se autentica con huella, se accede a las APIS nativas del teléfono para verificar la huella y habiendo logrado la verificación se genera un token en el server UAA para posteriores solicitudes

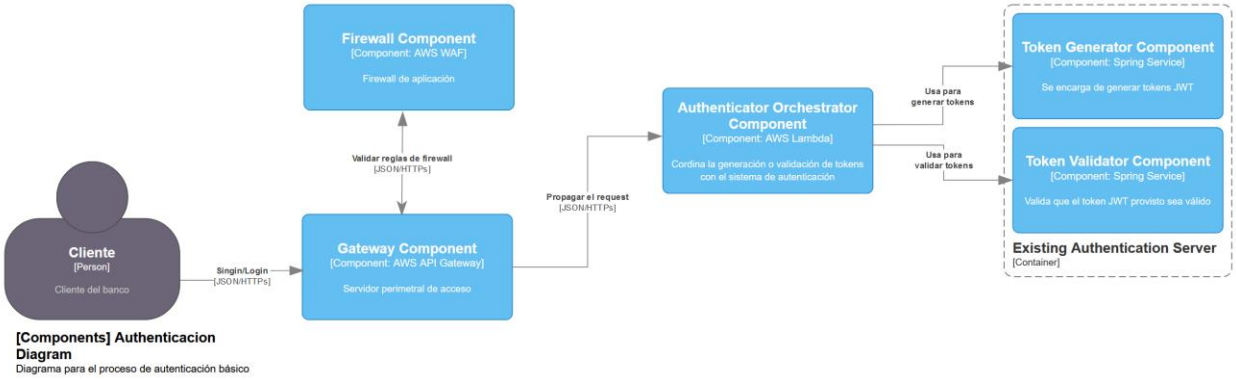
Observación 1: Por seguridad deberíamos configurar en el AWS WAF una **rate-based rule,** además deberíamos configurar **API Key y Usage Plans** para AWS API Gateway, todo esto a fin de limitar un posibles ataques de denegación de servicio

Observación 2: Podríamos agregar un AWS ClouFront, pero debemos desactivar el caché (en las configuraciones de la distribución y los headers como **Cache-Control**, **Exprires**) ya que al ser información sensible (datos de usuario, transferencias), no está permitido, por regulación (superintendencia de banca y seguros -SBS), almacenarlos en caché y menos en una caché perimetral, es decir, agregaríamos AWS CloudFront únicamente para protegernos de algunos ataques, pero no para distribuir contenido en caché.

Además, podemos suponer que al tratarse de un Banco Ecuatoriano no posee alcance global, es decir, no necesita distribuir contenido a nivel mundial; sin embargo, es posible aún usar CloudFront con "geographics restrictions" (allow list)

Observación 3: Podríamos usar AWS Shield Advance, pero este servicio (destinado principalmente a protegernos de ataques DDoS) tiene un costo bastante elevado.

Diagrama de componentes – C3



Proceso de Onboarding

Se nos ha dicho que ya tenemos un componente que implementa el protocolo OAuth2, a este componente lo hemos llamado "Authentication Server o Server UAA"; sin embargo, este componente no implementa la verificación biométrica fácil que será requerido para el proceso de registro. No sabemos si el componente anteriormente mencionado es extensible o si se trata de una caja negra que no podemos tocar, por esta razón hemos optado por implementar el proceso de onboarding con ayuda de AWS Rekognition y algunas funciones lambda.

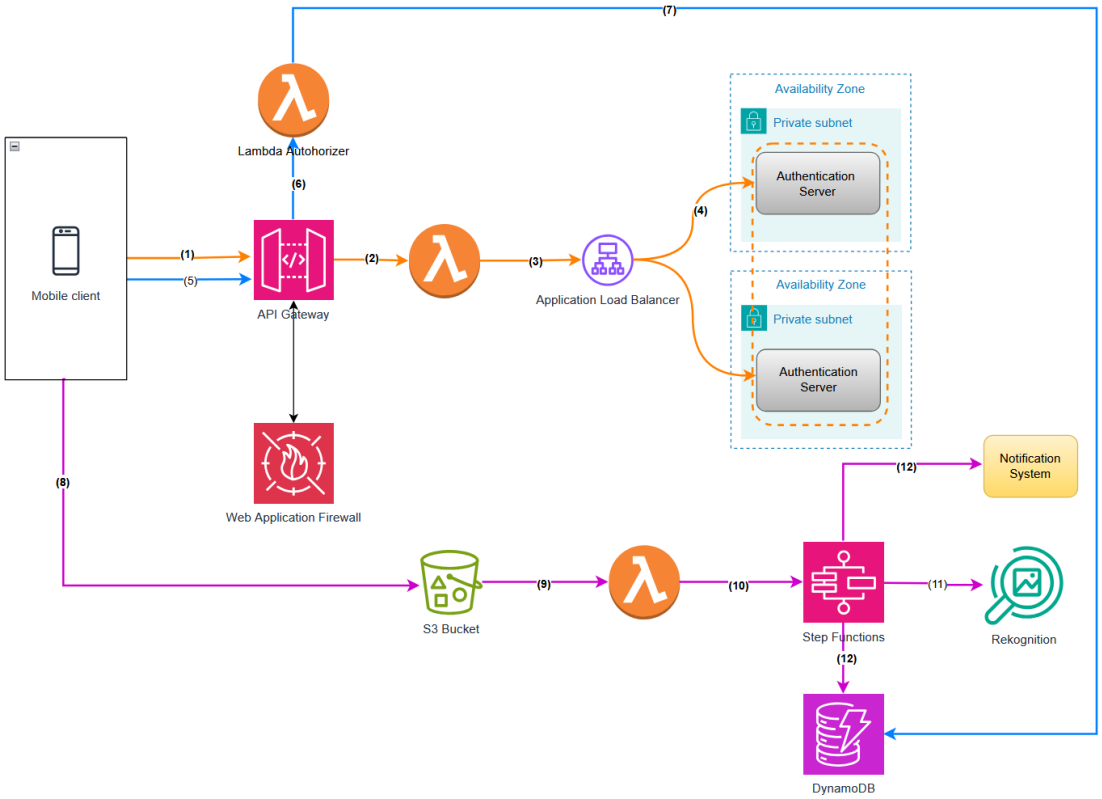
Explicación de pasos:

1. El usuario envía su formulario de registro a la URL: **/singin**

2. Los datos se propagan hacia una función lamba que hace algunas verificaciones, formateos y envía logs a CloudWatch
3. Se invoca el Authorization Server ya existente mediante el balanceador de carga (se colocó un balanceador de carga delante, junto con un grupo de AutoScaling en 2 AZ como mínimo para obtener HA)
4. Se registra el usuario y se genera un **Id-usuario**
5. El usuario ya con su **Password y Usuario** recién creados realiza una invocación a la URL **/login**
6. API Gateway se apoya de un Lambda Authorizer para realizar una autorización custom
7. Lambda Authorizer verifica en la base de datos DynamoBD (con el **Id-usuario**), al no encontrar datos, devuelve como respuesta una negativa de autorización y solicita cargar foto
8. El dispositivo móvil toma la foto mediante la API del teléfono y la sube a un bucket S3
9. Se dispara (trigger) una función lambda
10. Lambda desencadena una step function
11. Step Functions verifica con ayuda de AWS Rekognition si la imagen es válida (quizá le tomó foto a la pared)
12. Si la imagen no es válida, Step Function envía una notificación al teléfono del usuario indicando su error
13. Si la imagen sí es válida Step Functions registra los datos de la imagen junto con el id-usuario en DynamoDB

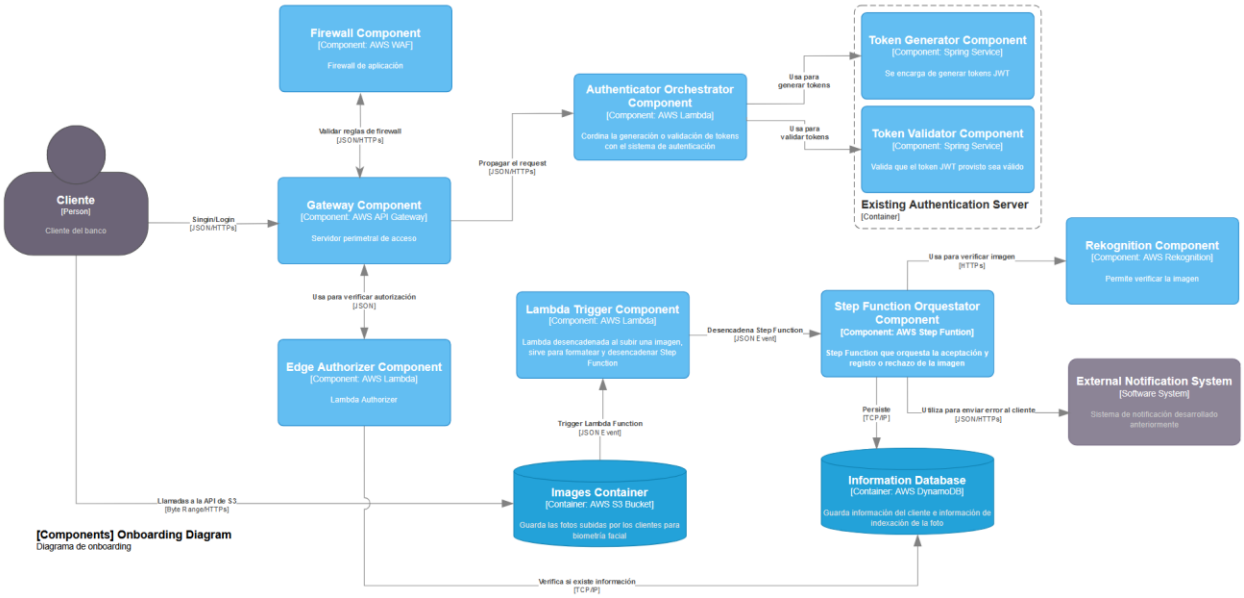
Observación: Debido a que solo se solicitó la biometría facial al momento del registro, la próxima que se inicie sesión con **usuario y password** el Lambda Authorizer verificará en DynamoDB que ya existe una imagen para ese usuario y por tanto dejará pasar la solicitud hacia el Authentication Server quien validará si el **usuario y password** son válidos o no. Si es que lo fuesen se generará un toque de acceso JWT, en caso contrario se devuelve al usuario un error de credenciales.

Diagrama de despliegue (implementación en AWS)



Como se puede ver, se trata de una ampliación del diagrama anterior de autenticación, únicamente le hemos “extendido” la funcionalidad de biometría facial para lograr el onboarding

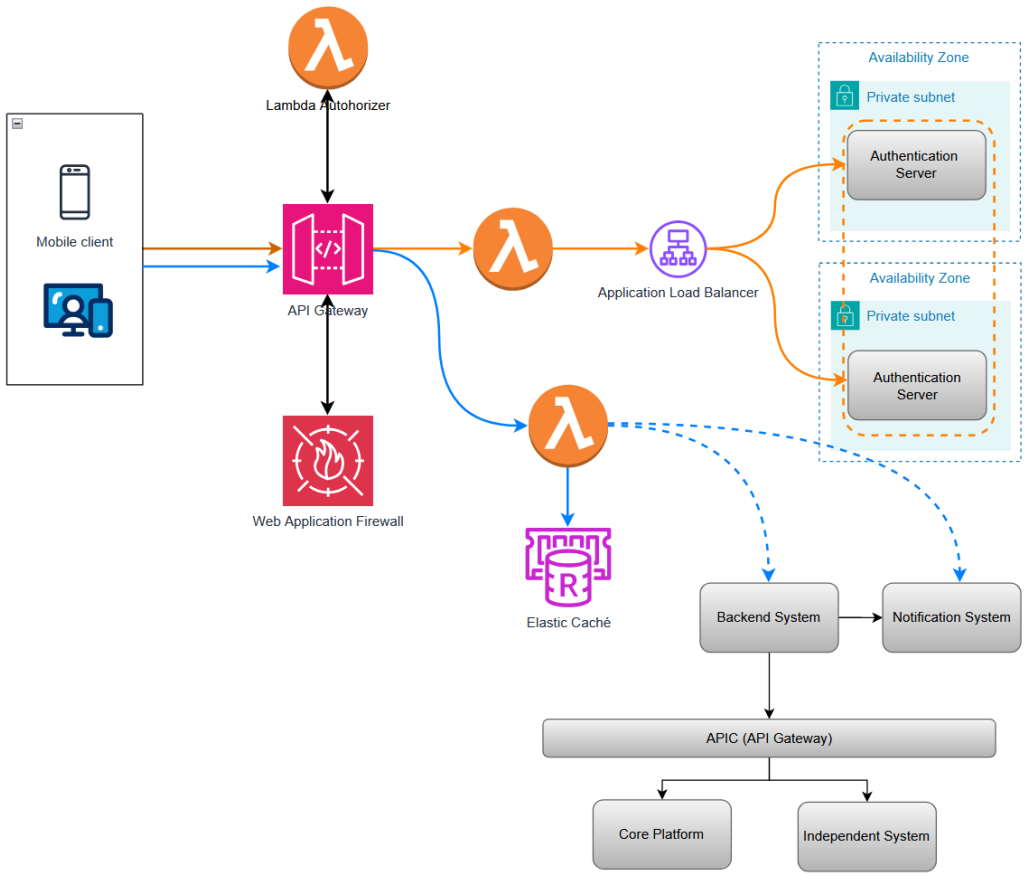
Diagrama de componentes – C3



Cedric Bardalez Hall

Observación: Sería interesante añadir algunos detalles de MFA, por ejemplo, podríamos pedir la confirmación por OTP para cada transferencia. Par lograr esto se envía un request al servicio encargado y el OTP generado se envía al usuario mediante el servicio de notificaciones previamente desarrollado, el OTP llega por SMS y debe introducirlo para confirmar la operación, luego el OTP se valida y se procede con la transferencia

Diagrama de despliegue OTP (implementación en AWS)



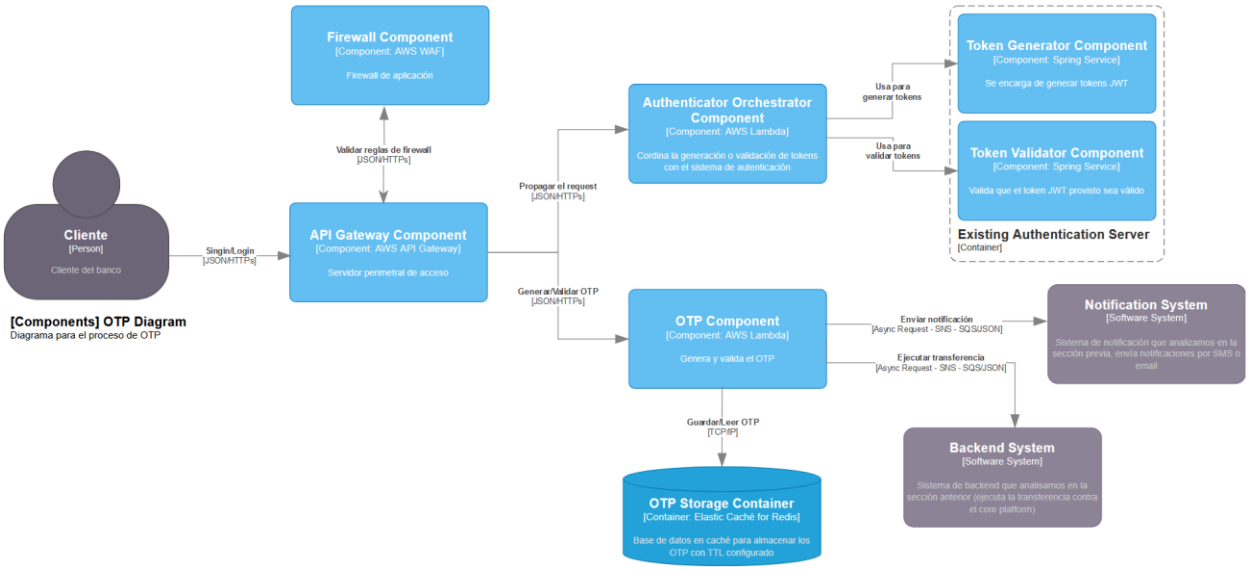
El flujo en naranja corresponde al proceso de autenticación habitual, explicado anteriormente, en el cual el usuario envía un request a **/login** junto con su password y usuario, la solicitud viaja hasta el componente de autenticación (Server UAA, ya existente) y de ser correctas las credenciales se retorna un token de autenticación JWT, en caso de error de credenciales se devolverá un "Invalid Credentials"

Una vez que el usuario ya se encuentra "logeado" en el sistema, querrá ejecutar una transferencia, y previo a ello se deberá generar y validar un OTP. Para logar esto, tenemos los siguientes pasos (ver flujo azul):

1. El sistema envía una solicitud al API Gateway **/otpGenerate**
2. La solicitud viaja a una función lambda que ejecuta un algoritmo y genera 4 dígitos con un tiempo de expiración variable (de acorde a configuración, pero por recomendación no debería exceder de unos pocos minutos, 2 ó 3).

3. Este número generado se **guarda en una memoria caché (asociado a los datos del usuario - token) y se configura el TTL** correspondiente
4. Posteriormente se invoca al servicio de notificación (desarrollado anteriormente). Notar que la línea es punteada porque es una invocación asíncrona, es decir se producirá enviando un mensaje a un tópico de SNS y este filtrará para "empujar" el mensaje a la cola correspondiente (ver sección de notificación).
5. El OTP es enviado al usuario quien deberá colocarlo en el formulario que aparece en su aplicación web o móvil
6. El sistema envía un nuevo request a **/otpValidate** con los datos del usuario (token) y los datos de la transferencia encriptados con llave pública.
7. Los datos anteriores llegan a la función lambda la cual valida (mediante la caché guardada anteriormente) si el OTP es válido, si corresponde al usuario y si aún no ha expirado.
8. Si se cumplen todas las condiciones, envía los datos de la transferencia, de forma asíncrona, al "Backend System"
9. Si existiera un error de validación se retorna la respuesta al usuario

Diagrama de componentes OTP – C3



Herramientas recomendadas para autenticación

Para el proceso de autenticación se podría implementar herramientas muy potentes como Okta o KeyClock. En general podemos decir que Okta es más potente que KeyClock, incluso tiene la posibilidad de elegir la cantidad y los tipos de factores a aplicar en un MFA, es decir de acorde a la ubicación del dispositivo (dirección IP), horario, patrones de acceso, riesgo de operación, etc. Solicitar desde un clave OTP de 4 dígitos hasta una biometría dactilar o facial, preguntas claves o incluso varios factores combinados; sin embargo, tiene un costo bastante elevado. En algunas ocasiones si es la primera vez que la empresa se acerca a este tipo de "external identity providers" quizá lo mejor sea empezar por algo más acotado como KeyClock que incluso es recomendado por el propio equipo de Spring y tiene un costo bastante menor.

También podríamos recomendar AWS Cognito, el cual manera un user-pool, y funciona para aplicaciones web y móviles, tiene la ventaja que es sencillo de configurar y trabaja perfectamente en el entorno de AWS que ya estamos implementado. Además, posee la capacidad de federación mediante un "indentity provider externo" y "social- authentication" mediante Facebook, Twitter, Google, etc., es decir, incluso podría seguir usando las capacidades de nuestro componente (server UAA) que ya lo tenemos construido (nuestro componente actuaría como external identity provider).

Estrategia de Distaster Recovery (DRP)

Para DRP tenemos 4 posibles estrategias:

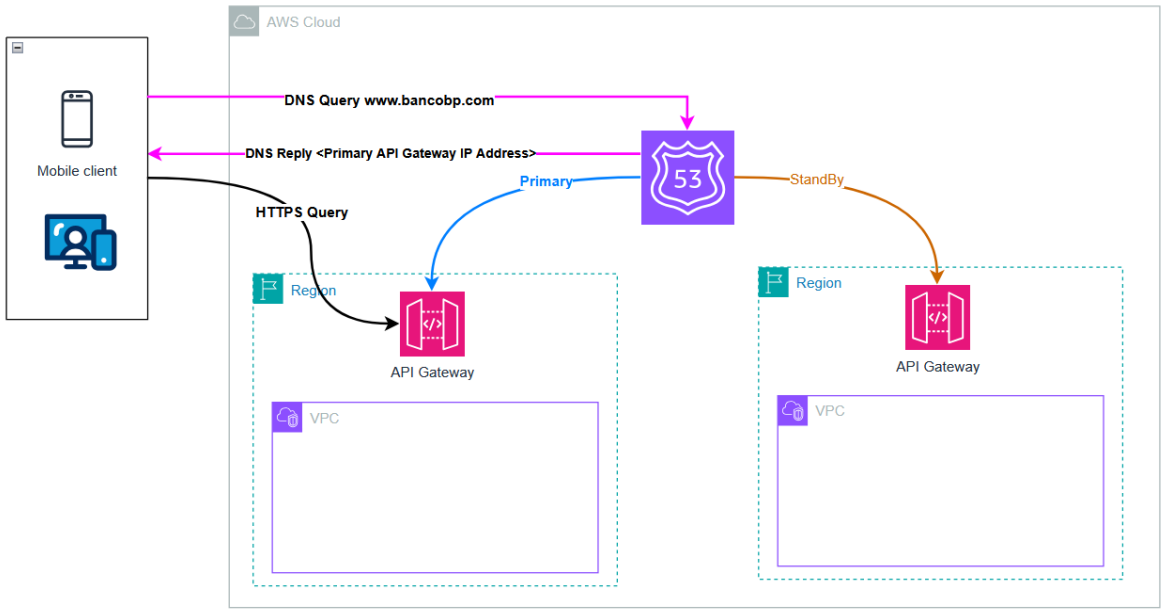
- Backup and Restore (RPO horas, RTO horas)
- Pilot Light (RPO minutos, RTO horas)
- Warm Standby (RPO segundos, RTO minutos)
- Activo-Activo (RPO segundo/instantáneo, RTO segundo)

Evidentemente usar una estrategia tipo **Backup and Restore** nos resulta más económico, pero puede tener un RPO y RTO de horas, en ninguna parte del ejercicio indica requisitos explícitos de RTO y RPO; sin embargo, al tratarse de una aplicación bancaria puedo suponer que no deberíamos tener un downtime mayor a unos pocos minutos.

Notar también que, bajo la arquitectura supuesta, no poseemos la data en nuestro dominio (nuestros servicios a desarrollar no son propietarios de la data), es decir, la data la maneja la **"plataforma core"** y el **"sistema independiente"** los cuales nos permiten hacer transferencias y obtener información del cliente, es decir para replicar nuestra infraestructura a nivel nube lo único que necesitamos es redesplegar servidores de aplicaciones (en el caso más extremo en que toda la región de AWS sufriera un desperfecto).

Por los motivos expuestos usaremos una estrategia de **Warm standby** manteniendo servidores con una carga mínima en otra región (necesitamos únicamente escalar en caso de conmutar de región) y esta infraestructura la automatizamos con CloudFormation o AWS SDK

Observación 1: Usaremos el servicio de AWS Route 53 con una estrategia de failover, mediante el cual podemos rápidamente conmutar a la otra región



Observación 2: También podríamos usar el servicio de AWS Elastic Disaster Recovery, que nos ayuda en esta labor, manteniendo un RPO de segundos y RTO de minutos (como si fuese un Warm Standby) con un costo tipo Pilot Light.

El servicio en mención logra este desempeño al utilizar el concepto de "stagging area", es decir copia la dada y los servicio a un área reservada por AWS en una región indicada. Esta copia posea un storage de bajo costo y una capacidad de cómputo mínima (optimizado por AWS). Si existiera un desastre, solo se promueven los elementos de esta área

No optamos por esta solución, a pesar de sus beneficios, porque estamos empezando nuestra aplicación y no deseamos amarrar nuestra estrategia de DRP a AWS. Se considera los planes de DRP más genérico, aplicables a múltiples entornos cloud.

Observación: Para complementar nuestra estrategia de DRP, debemos considerar lo siguiente

- El S3 bucket con la foto de los usuarios (proceso de onboarding) deberá ser replicado a la otra región mediante un proceso de cross-replication
- Para la información sobre la foto y el usuario (almacenados en DynamoDB), usaremos global tables de DynamoDB para replicarlo a la otra región.

Cacheando información de usuario para reducir latencia

Usaremos AWS Elastic cache y existen 3 posibles estrategias de caché:

- TTL
- Write Through
- Lazy loading

Lo único que vamos a cachear será la información **básica** del usuario, para reducir latencia y evitar llamadas a on-premise, esta información será cargada únicamente tras ser solicitada (**Lazy Loading),** además deberá ser encriptada con SSE.

Cedric Bardalez Hall

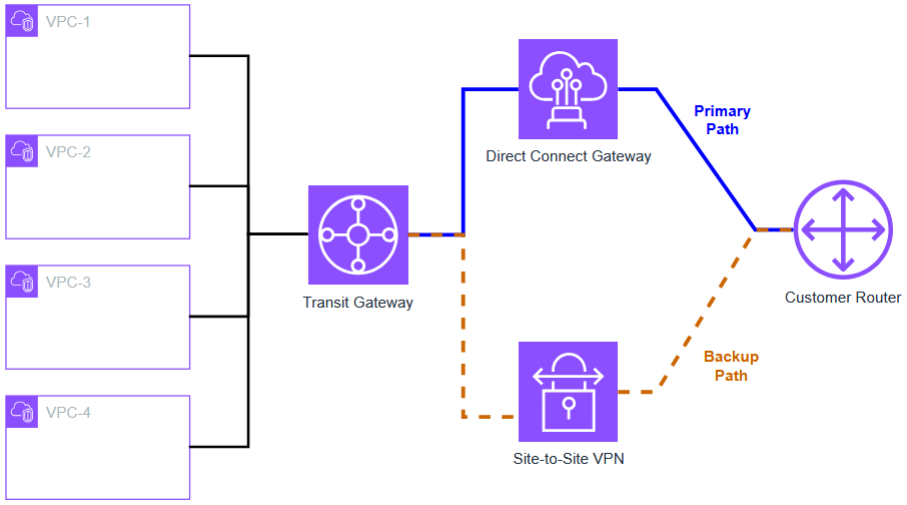
Sistemas ya existentes

Observación: Notamos que el **sistema independiente** tiene información completa del usuario, en cambio la **plataforma core** solo tiene datos básicos, podríamos suponer que estos sistemas son cajas negras, que no debemos tocar, y ellos de alguna manera (con algún mecanismo) logran sincronizar sus datos y propagar actualizaciones.

Consideraciones de acceso a los servicios ya existentes

No sabemos si los sistemas ya existentes (plataforma core y sistema independiente) están en on-premise, AWS u otro Cloud Provider, pero para retarnos un poquito más supondremos que se encuentran en on-premise y necesitamos tener una conectividad HA de nuestra AWS VPC a on-premise network y el API Gateway mencionado en el ejercicio no sería el servicio de AWS API Gateway si no que haría referencia a un servicio genérico tipo IBM API Connect

Para lograr una conectividad en HA usaremos el siguiente esquema de conexión:



Consideraciones sobre la ejecución de transferencias

Observación: Existe un gran problema que debemos tratar, que pasaría si el servicio ejecuta una solicitud POST hacia la Core Platform (para realizar una transferencia), la Core Platform responde que la transferencia se realizó correctamente y cuando nuestro servicio está a punto de enviar la notificación hacia SNS sufre una caída abrupta (se generaría una transacción nunca fue notificada)

Para solucionar este problema usaremos una pequeña base de datos AWS RDS MySQL en la cual manejaremos los siguientes estados.

Para la transferencia:

- EVENT_CONSUMED
- REQUEST_BEFORE
- WAITING
- SUCCESS

- FAIL

Y para la notificación:

- NULL
- NOTIFY
- NOTIFICATED

Como primer paso consumiremos el mensaje de la cola SQS y verificaremos si ya existe un registro con dicho **Event-Id**, en caso de no existir guardaremos los siguientes datos en nuestra base de datos:

Event-Id	Status Transfer	Details	Status Notification
2040506	EVENT_CONSUMED	El evento fue consumido	NULL

Si ocurriese un error no recuperable, y nuestro servicio se cae, debido a **visibilityTimeOut** de la cola, el mensaje estará disponible para ser consumido nuevamente por otra aplicación, la cual verificará en base de datos y al encontrar el registro previo actualizará el campo **Details** para reportar el error y también actualizará el campo **Status Transfer** con el valor REQUEST_BEFORE

Event-Id	Status Transfer	Details	Status Notification
2040506	REQUEST_BEFORE	Hubo error en EVENT_CONSUMED	NULL

Si el sistema se cayese en este preciso momento, una nueva instancia, debido a **visibilityTimeOut** de la cola, consumirá nuevamente el mensaje y verificará la base de datos. Al encontrar el **Status Transfer** en REQUEST_BEFORE, procederá a enviar una solicitud al Core Platform solicitando los últimos movimientos, para verificar si el sistema se cayó antes de enviar el request o justo después de enviar el request y antes de cambiar el **Status Transfer** a WATING en base de datos, de no haber ejecutado el request va a enviar la solicitud de transferencia al Core Platform y actualizará el campo **Status Transfer** de base de datos a WAITING

Event-Id	Status Transfer	Details	Status Notification
2040506	WAITING	Hubo error en EVENT_CONSUMED	NULL

Observación: El paso anterior parece absurdo, podríamos pensar el estado EVENT_CONSUMED y REQUEST_BEFORE son lo mismo. Si he consumido el evento obviamente estoy previo a ejecutar el request de transferencia; sin embargo, debemos recordar que se pueden consumir hasta 10 mensajes en un solo pull, pero no se puede enviar 10 transferencias en batch para la Core Platform. En otras palabras, se consumirán 10 mensajes y se guardaran 10 registros en base de datos con el Status Transfer EVENT_CONSUMED luego se ejecutará el primer request hacia la Core Platform, cambiando el Status Transfer a REQUEST_BEFORE para uno de los registros.

Si el sistema se cayese en este preciso momento, una nueva instancia, debido a **visibilityTimeOut** de la cola, consumirá nuevamente el mensaje y verificará la base de datos. Al encontrar el **Status Transfer** en WAITING enviará una solicitud al Core Platform solicitando los últimos movimientos, para verificar si se ejecutó correctamente, en caso negativo reintentará la operación de transferencia hasta en 3 ocasiones (para errores reintentables) y posteriormente actualizará el campo **Status Transfer** a FAIL y en caso positivo actualizará el campo **Status Transfer** de la base de datos a SUCCED y el campo **Satus Notification** a NOTIFY

Event-Id	Status Transfer	Details	Status Notification
2040506	SUCCESS/FAIL	Hubo error en EVENT_CONSUMED	NOTIFY

Si el sistema se cayese en este momento, no existiera problema alguno, ya que la nueva instancia consumirá nuevamente el evento al bróker de mensajes SQS y al verificar en la base de datos notaría que únicamente falta notificar.

Si el sistema envía la notificación al tópico SNS y se cayese antes de actualizar la base de datos, no existiría problema ya que el nuevo servicio que tome el mensaje va a leer el estado de la base de datos y enviará nuevamente la notificación (como hemos visto anteriormente, no es un problema grave notificar 2 veces con el mismo id).

Finalmente, el sistema actualiza el campo **Stratus Notification** de la base de datos al valor NOTIFICATED y avisará al bróker que el mensaje fue procesado correctamente, para ser eliminado definitivamente de la cola.

Observación: Todo el flujo detallado anteriormente, nos permite tener doble o “n veces” de notificación, pero solo una transferencia ejecutada.

Observación: Por supuesto que manejaremos reintentos (para errores re-intentables) con un back-off exponencial hasta un número de veces que negocio considere conveniente y definidos mediante variables de configuración.

Observación: Debemos identificar las operaciones idempotentes y los errores rentintables.

- Consulta de información básica de cliente – Idempotente – Plataforma Core
- Consulta de transferencias – Idempotente – Plataforma Core
- Consulta información completa de cliente – Idempotente – Sistema independiente
- Transferencia – No idempotente - Reintentable para errores 5XX, TimeOut.

Consideraciones de encriptación y seguridad

Observación: Podría considerarse por gusto las aplicaciones backend java y pensar que al eliminarlas e ir directo al API Gateway on-premise (API Connect) estamos reduciendo latencia, lo cual es cierto; sin embargo, esto sería riesgoso. Debemos practicar **defensa en profundidad**, de manera que, si se vieran comprometidas las aplicaciones web, tenemos una segunda capa de servidores que formatean, traducen y filtran data, orquestan operaciones, agregan y componen información.

Consideraciones adicionales:

- Toda la data almacenada debe estar encriptada (en nuestro caso, los datos en cache, los datos de la foto del usuario y la propia foto). Para ello se usará las capacidades de DynamoDb, Elastic Caché y S3 como Sever Side Encryption.
- No es necesario encriptar la data del estado de la transferencia.
- Le herramienta KMS administrará las claves de usuario KMS-C y estas deberán configurar la rotación obligatoria y la replicación entre regiones.
- La replicación de llave a otra región se pide para el proceso de DRP y no aumentar la latencia yendo a otra región.
- Las configuraciones de las aplicaciones se almacenan en AWS SSM Parameter Store y bajo ninguna consideración se permitirá usar configuraciones en el propio servidor de aplicación
- El acceso a los servidores, para configuración o instalaciones no está permitido, todo debe estar automatizado en despliegues CloudFormation, para mantener un historial de cambios e identificar drifts.
- De ser necesario y debidamente justificado se permitirá el acceso al servidor únicamente mediante el AWS SSM Sesión Manager y dicha sesión será auditada y los logs almacenados en un bucket de S3.
- Las credenciales, API Keys serán almacenadas en AWS Secret Manager y replicadas a la región de DRP. Estas credenciales deben tener la rotación automática activada.
- Se utilizará activamente AWS CloudTrail para verificar las acciones realizadas por los usuarios y administradores que tienen acceso a AWS.
- Para los acceso se usará la estrategia de mínimos privilegios en IAM, además de una agrupación por grupos para poder otorgar roles. Solo bajo circunstancias muy especiales se permitirá adjuntar una política inline directamente al usuario.
- Toda solicitud debe viajar mediante TLS, por tanto, debemos generar certificados privados mediante AWS Private Certificate Authority.
- Para las solicitudes exteriores no se permitirá viajar por protocolo no seguro, es decir se debe redireccionar al puerto 443 y colocar el certificado correspondiente en API Gateway emitido mediante AWS Certificate Manager.
- Los certificados emitidos por AWS Certificate Manager tendrán una duración de 13 meses y se deberá alertar de su vencimiento con un mínimo de 35 días antes, configurados mediante AWS Config con la regla acm-certificate-expiration-check.
- Toda llamada entre servicios debe viajar por tráfico interno, por tanto, utilizaremos repetidamente el servicio de PrivateLink.
- Se deberá mantener alertas en AWS Health para recibir notificaciones sobre posibles fallos en AWS que afecten nuestros servicios.
- Toda instancia ec2 (frontend y backend) y acceso a servicio (RDS, VPC Endpoints, etc.) debe configurar un security group adecuadamente.

Consideraciones sobre la auditoria

- **Nivel administración:** Usaremos SSM Sesión Manager y volcamos los logs a S3, luego usamos AWS Athena para hacer consultas one-time sobre dichos registros (registros de auditoria de ingreso al servidor)

- **Nivel de usuarios:** Recolectaremos datos mediante eventos, tal como se explica a continuación:

Ya que tenemos colas SQS para la comunicación, podemos agregar una cola extra y un lambda colector que guarda la data en DynamoDB y sirve como base de datos de auditoría de todas las operaciones realizadas por los usuarios.

Incluso podrimos llevar toda esta data a OpenSearch, **usando un OpenSearch Ingestion pipeline con Amazon DynamoDB** y al ser una base de datos indexada podemos hacer búsquedas por usuario, por fecha, etc. o incluso podríamos levantar un Dashboard de visualización

Diagrama de despliegue (implementación en AWS)

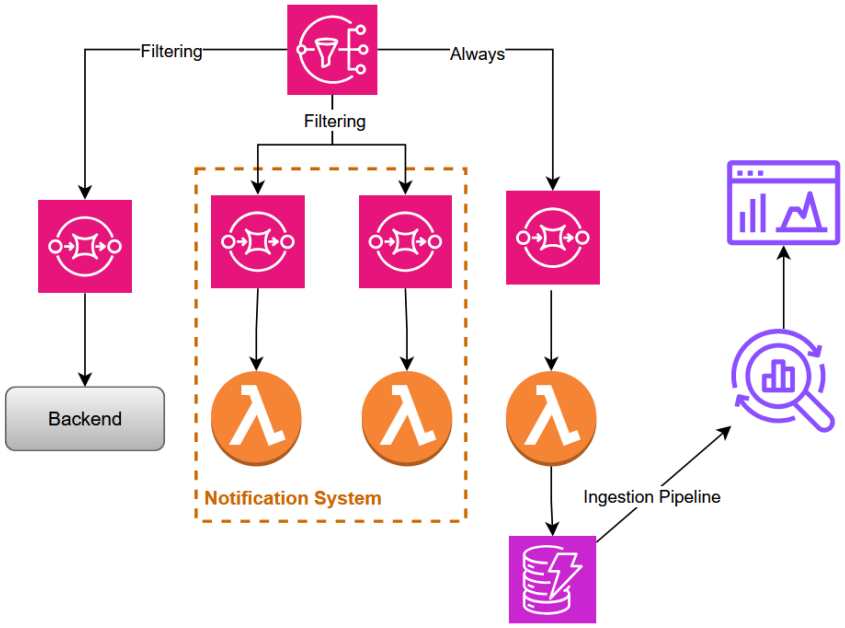
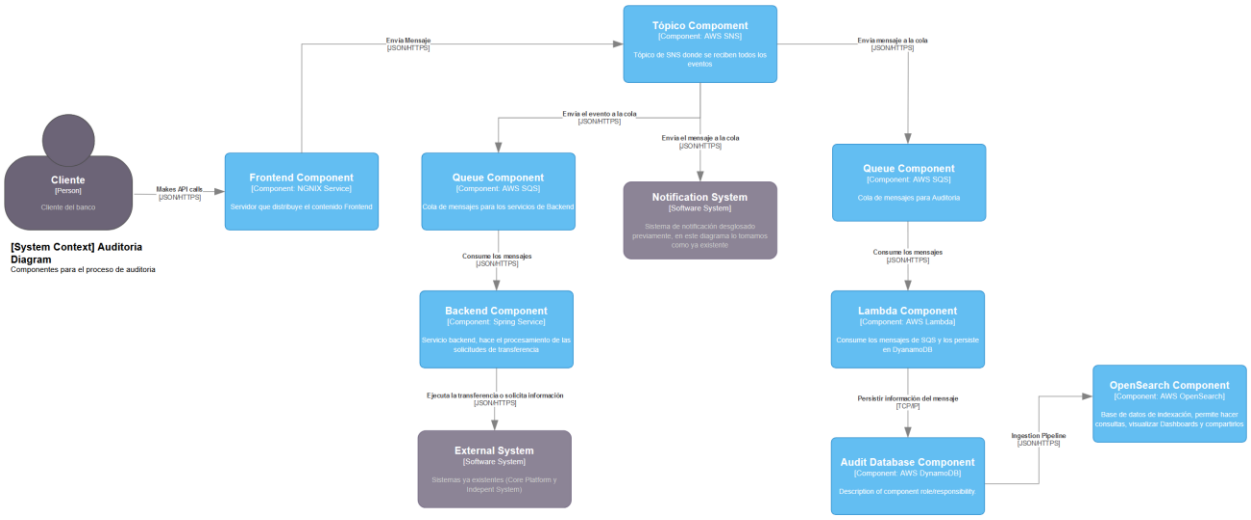


Diagrama de componentes – C3



Consideraciones sobre monitoreo

Usaremos AWS CloudWatch y AWS Config

- **AWS Config:** Lo usaremos para notificar el vencimiento de los certificados ACM, existe una regla llamada **acm-certificate-expiration-check** (configurable número de días), e incluso podemos enviar una notificación con SNS al correo del administrador de sistemas.
- En CloudWatch generaremos dashboards con las métricas que deseamos y crearemos algunas métricas relevantes para nuestro uso, como el consumo promedio de CPU, el número de operaciones I/O, el consumo promedio de uso de memoria (para esta métrica debemos configurar el agente en los servidores)

Observación: Para el monitoreo utilizaremos la solución de CloudWatch Logs Group, y para la trazabilidad X-Ray, en servidores EC2 instalaremos el agente y la trazabilidad y logs de las funciones lamba viene casi de regalo gracias al execution-role adjunto.